

Tutorial of the
2004 Pascal Workshop
On Text Understanding and Mining

Kernel Methods for Textual Data
– Application to NLP Problems

Jean-Michel Renders
Xerox Research Center Europe

27 January 2002
XRCE, Grenoble
France

Acknowledgments

- This tutorial is based both:
 - On a tutorial given at ACL-2002 (Philadelphia)
 - On material gleaned from the book

Kernel Methods for Pattern Analysis

By J. Shawe-Taylor, N. Christianini

...to be published very soon

Many thanks to John for sending me the draft version!

- It largely relies on the EU Kermit Project

Agenda

- What's the philosophy of Kernel Methods?
- How to use Kernels Methods in Learning tasks?
- Kernels for text (BOW, latent concept, string, word sequence, tree and Fisher Kernels)
- Applications to NLP tasks

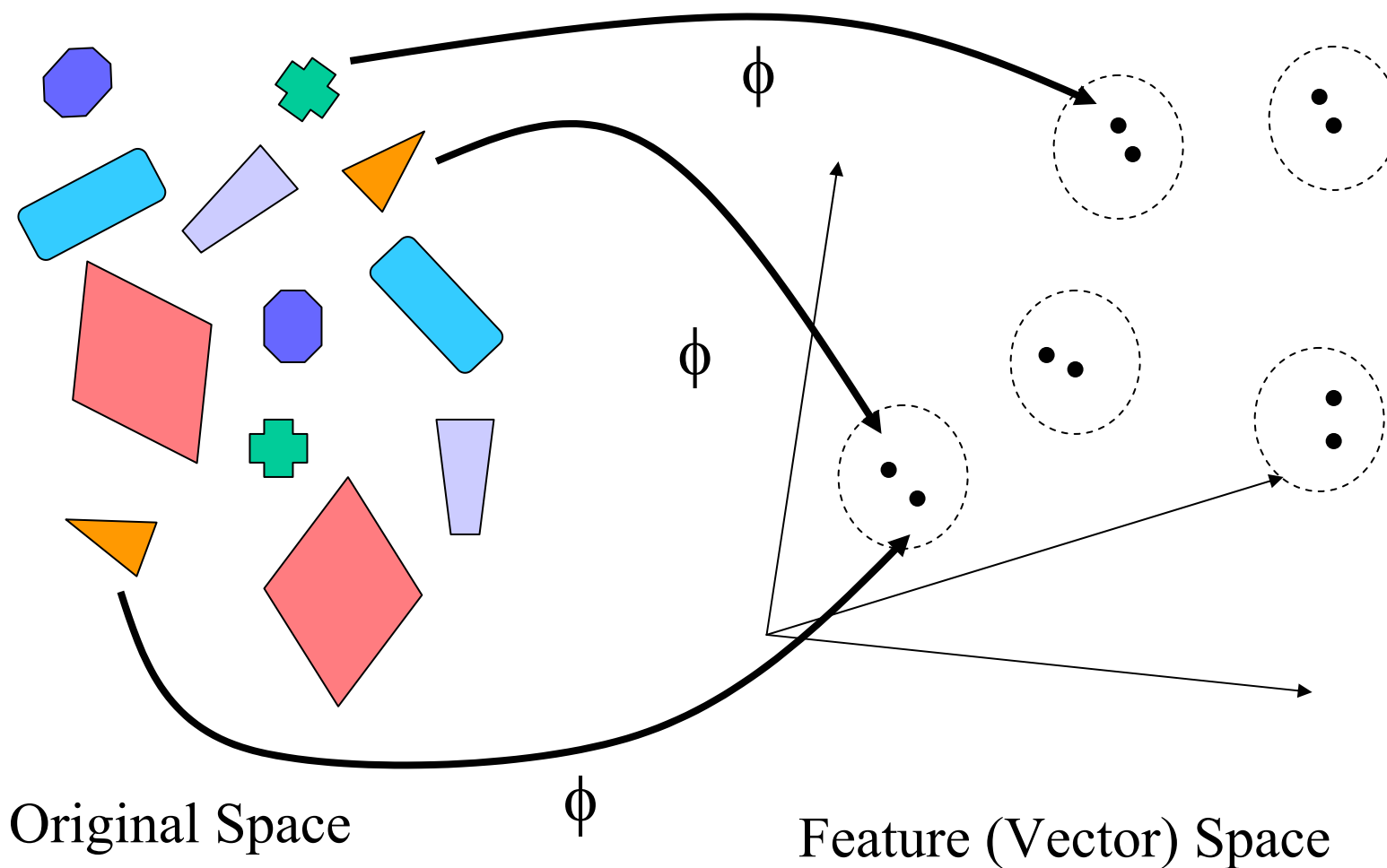
Plan

- What's the philosophy of Kernel Methods?
- How to use Kernels Methods in Learning tasks?
- Kernels for text (BOW, latent concept, string, word sequence, tree and Fisher Kernels)
- Applications to NLP tasks

Kernel Methods : intuitive idea

- Find a mapping ϕ such that, in the new space, problem solving is easier (e.g. linear)
- The *kernel* represents the similarity between two objects (documents, terms, ...), defined as the dot-product in this new vector space
- But the mapping is left implicit
- Easy generalization of a lot of dot-product (or distance) based pattern recognition algorithms

Kernel Methods : the mapping



Kernel : more formal definition

■ A kernel $k(x,y)$

- is a similarity measure
- defined by an implicit mapping ϕ ,
- from the original space to a vector space (feature space)
- such that: $k(x,y)=\phi(x)\cdot\phi(y)$

■ This similarity measure and the mapping include:

- Invariance or other a priori knowledge
- Simpler structure (linear representation of the data)
- The class of functions the solution is taken from
- Possibly infinite dimension (hypothesis space for learning)
- ... but still computational efficiency when computing $k(x,y)$

General Principles
governing Kernel Design

Benefits from kernels

- Generalizes (nonlinearly) pattern recognition algorithms in clustering, classification, density estimation, ...
 - When these algorithms are dot-product based, by replacing the dot product $(x \cdot y)$ by $k(x,y) = \phi(x) \cdot \phi(y)$
e.g.: linear discriminant analysis, logistic regression, perceptron, SOM, PCA, ICA, ...
NM. This often implies to work with the “dual” form of the algo.
 - When these algorithms are distance-based, by replacing $d(x,y)$ by $k(x,x) + k(y,y) - 2k(x,y)$
- Freedom of choosing ϕ implies a large variety of learning algorithms

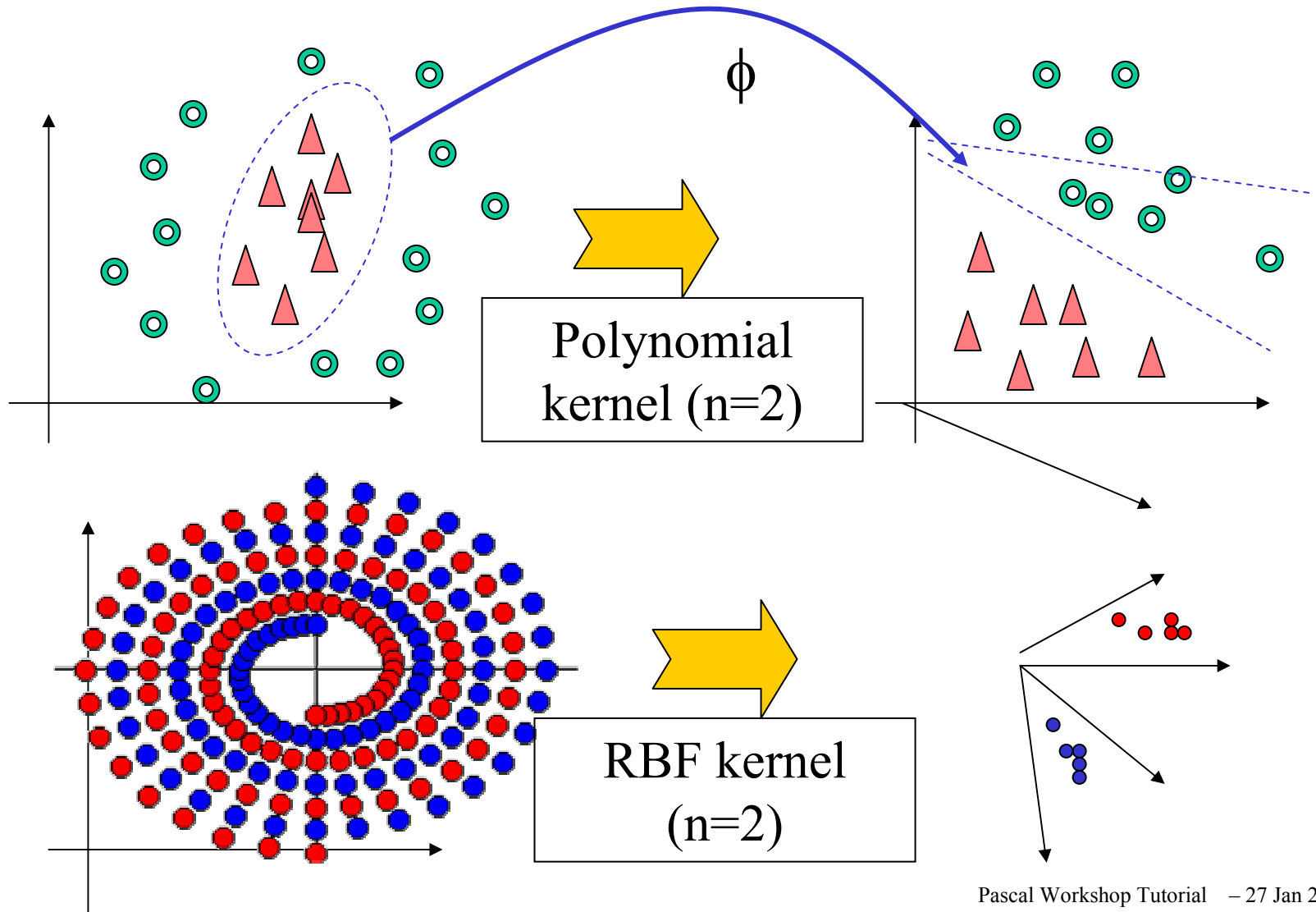
Valid Kernels

- The function $k(x,y)$ is a valid kernel, if there exists a mapping ϕ into a vector space (with a dot-product) such that k can be expressed as $k(x,y)=\phi(x)\cdot\phi(y)$
- Theorem: $k(x,y)$ is a valid kernel if k is positive definite and symmetric (Mercer Kernel)
 - A function is P.D. if $\int K(\mathbf{x},\mathbf{y})f(\mathbf{x})f(\mathbf{y})d\mathbf{x}d\mathbf{y} \geq 0 \quad \forall f \in L_2$
 - In other words, the Gram matrix \mathbf{K} (whose elements are $k(x_i,x_j)$) must be positive definite for all x_i, x_j of the input space
 - One possible choice of $\phi(x)$: $k(\cdot,x)$ (maps a point x to a function $k(\cdot,x) \rightarrow$ feature space with infinite dimension!)

Example of Kernels (I)

- Polynomial Kernels: $k(x,y)=(x\cdot y)^d$
 - Assume we know most information is contained in monomials (e.g. multiword terms) of degree d (e.g. $d=2$: x_1^2, x_2^2, x_1x_2)
 - Theorem: the (implicit) feature space contains all possible monomials of degree d (ex: $n=250$; $d=5$; $\dim F=10^{10}$)
 - But kernel computation is only marginally more complex than standard dot product!
 - For $k(x,y)=(x\cdot y+1)^d$, the (implicit) feature space contains all possible monomials up to degree d !

Examples of Kernels (III)



The Kernel Gram Matrix

- With KM-based learning, the **sole** information used from the training data set is the Kernel Gram Matrix

$$K_{training} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \dots & k(\mathbf{x}_1, \mathbf{x}_m) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \dots & k(\mathbf{x}_2, \mathbf{x}_m) \\ \dots & \dots & \dots & \dots \\ k(\mathbf{x}_m, \mathbf{x}_1) & k(\mathbf{x}_m, \mathbf{x}_2) & \dots & k(\mathbf{x}_m, \mathbf{x}_m) \end{bmatrix}$$

- If the kernel is valid, K is symmetric definite-positive .

The right kernel for the right task

- Assume a categorization task: the ideal Kernel matrix is
 - $k(\mathbf{x}_i, \mathbf{x}_j) = +1$ if \mathbf{x}_i and \mathbf{x}_j belong to the same class
 - $k(\mathbf{x}_i, \mathbf{x}_j) = -1$ if \mathbf{x}_i and \mathbf{x}_j belong to different classes
 - \rightarrow concept of target alignment (adapt the kernel to the labels), where alignment is the similarity between the current gram matrix and the ideal one [“two clusters” kernel]
- A certainly bad kernel is the diagonal kernel
 - $k(\mathbf{x}_i, \mathbf{x}_j) = +1$ if $\mathbf{x}_i = \mathbf{x}_j$
 - $k(\mathbf{x}_i, \mathbf{x}_j) = 0$ elsewhere
 - All points are orthogonal : no more cluster, no more structure

How to choose kernels?

- There is no absolute rules for choosing the right kernel, adapted to a particular problem
- Kernel design can start from the desired feature space, from combination or from data
- Some considerations are important:
 - Use kernel to introduce a priori (domain) knowledge
 - Be sure to keep some information structure in the feature space
 - Experimentally, there is some “robustness” in the choice, if the chosen kernels provide an acceptable trade-off between
 - simpler and more efficient structure (e.g. linear separability), which requires some “explosion”
 - Information structure preserving, which requires that the “explosion” is not too strong.

How to build new kernels

■ Kernel combinations, preserving *validity*:

$$K(\mathbf{x}, \mathbf{y}) = \lambda K_1(\mathbf{x}, \mathbf{y}) + (1 - \lambda) K_2(\mathbf{x}, \mathbf{y}) \quad 0 \leq \lambda \leq 1$$

$$K(\mathbf{x}, \mathbf{y}) = a \cdot K_1(\mathbf{x}, \mathbf{y}) \quad a > 0$$

$$K(\mathbf{x}, \mathbf{y}) = K_1(\mathbf{x}, \mathbf{y}) \cdot K_2(\mathbf{x}, \mathbf{y})$$

$$K(\mathbf{x}, \mathbf{y}) = f(x) \cdot f(y) \quad f \text{ is real - valued function}$$

$$K(\mathbf{x}, \mathbf{y}) = K_3(\boldsymbol{\varphi}(\mathbf{x}), \boldsymbol{\varphi}(\mathbf{y}))$$

$$K(\mathbf{x}, \mathbf{y}) = \mathbf{x}' P \mathbf{y} \quad P \text{ symmetric definite positive}$$

$$K(\mathbf{x}, \mathbf{y}) = \frac{K_1(\mathbf{x}, \mathbf{y})}{\sqrt{K_1(\mathbf{x}, \mathbf{x})} \sqrt{K_1(\mathbf{y}, \mathbf{y})}}$$

Kernels built from data

- In general, this mode of kernel design can use both labeled and unlabeled data of the training set! Very useful for semi-supervised learning
- Basic ideas:
 - Intuitively, kernels define clusters in the feature space, and we want to find interesting clusters, i.e. cluster components that can be associated with labels.
 - Convex linear combination of kernels in a given family: find the best coefficient of eigen-components of the (complete) kernel matrix by maximizing the alignment on the labeled training data.
 - Build a generative model of the data, then use the Fischer Kernel (see later)

Kernels for texts

Increased use of syntactic and semantic info



■ Similarity between documents?

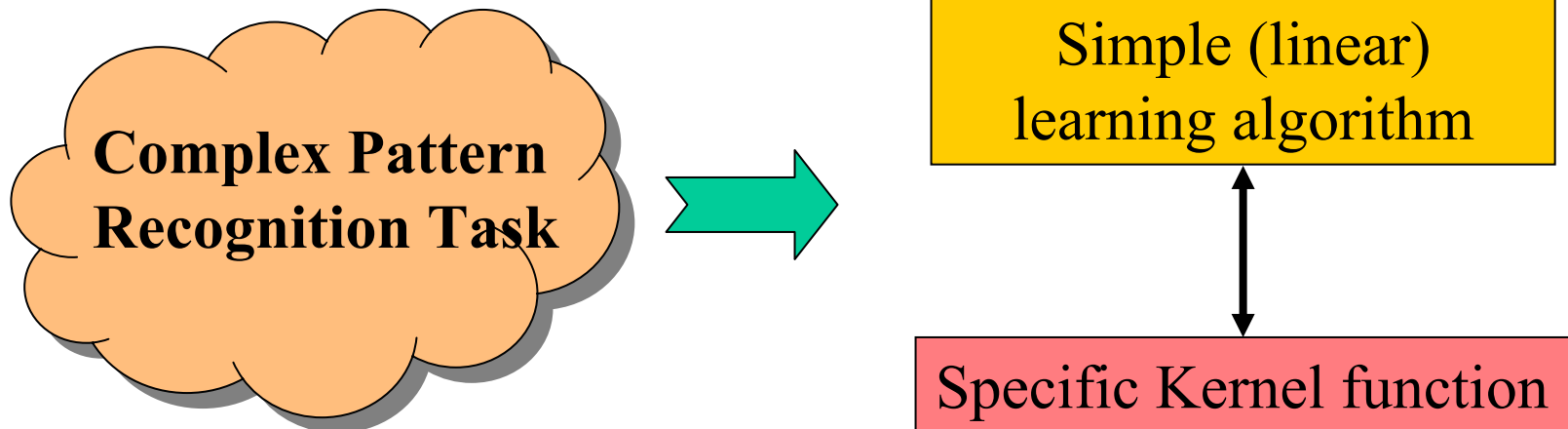
- Seen as 'bag of words' : dot product or polynomial kernels (multi-words)
- Seen as set of concepts : GVSM kernels, Kernel LSI (or Kernel PCA), Kernel ICA, ...possibly multilingual
- Seen as string of characters: string kernels
- Seen as string of terms/concepts: word sequence kernels
- Seen as trees (dependency or parsing trees): tree kernels
- Etc.

Agenda

- What's the philosophy of Kernel Methods?
- How to use **Kernels Methods** in Learning tasks?
- Kernels for text (BOW, latent concept, string, word sequence, tree and Fisher Kernels)
- Applications to NLP tasks

Kernels and Learning

- In Kernel-based learning algorithms, problem solving is now decoupled into:
 - A general purpose learning algorithm (e.g. SVM, PCA, ...) – Often linear algorithm (well-funded, robustness, ...)
 - A problem specific kernel

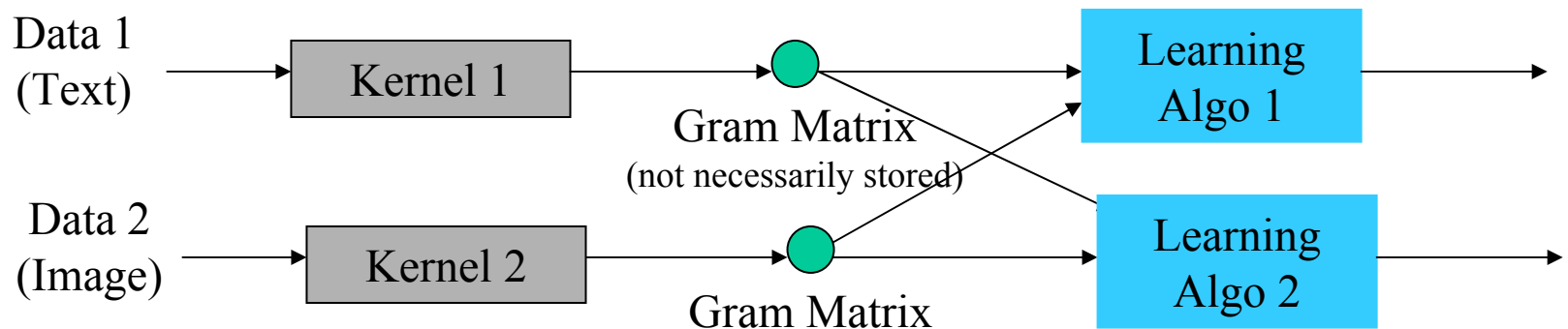


Learning in the feature space: Issues

- High dimensionality allows to render flat complex patterns by “explosion”
 - Computational issue, solved by designing kernels (efficiency in space and time)
 - Statistical issue (generalization), solved by the learning algorithm and also by the kernel
 - e.g. SVM, solving this complexity problem by maximizing the margin and the dual formulation
- E.g. RBF-kernel, playing with the σ parameter
- With adequate learning algorithms and kernels, high dimensionality is no longer an issue

Current Synthesis

- Modularity and re-usability
 - Same kernel ,different learning algorithms
 - Different kernels, same learning algorithms
- This tutorial is allowed to focus only on designing kernels for textual data



Agenda

- What's the philosophy of Kernel Methods?
- How to use Kernels Methods in Learning tasks?
- Kernels for text (BOW, latent concept, string, word sequence, tree and Fisher Kernels)
- Applications to NLP tasks

Kernels for texts

■ Similarity between documents?

- Seen as 'bag of words' : dot product or polynomial kernels (multi-words)
- Seen as set of concepts : GVSM kernels, Kernel LSI (or Kernel PCA), Kernel ICA, ...possibly multilingual
- Seen as string of characters: string kernels
- Seen as string of terms/concepts: word sequence kernels
- Seen as trees (dependency or parsing trees): tree kernels
- Seen as the realization of probability distribution (generative model)

Strategies of Design

- Kernel as a way to encode prior information
 - Invariance: synonymy, document length, ...
 - Linguistic processing: word normalisation, semantics, stopwords, weighting scheme, ...
- Convolution Kernels: text is a recursively-defined data structure. How to build “global” kernels from local (atomic level) kernels?
- Generative model-based kernels: the “topology” of the problem will be translated into a kernel function


Strategies of Design



- Kernel as a way to encode prior information

- Invariance: synonymy, document length, ...

- Linguistic processing: word normalisation, semantics, stopwords, weighting scheme, ...



- Convolution Kernels: text is a recursively-defined data structure. How to build “global” kernels from local (atomic level) kernels?



- Generative model-based kernels: the “topology” of the problem will be translated into a kernel function

'Bag of words' kernels (I)

- Document seen as a vector \mathbf{d} , indexed by all the elements of a (controlled) dictionary. The entry is equal to the number of occurrences.
- A training corpus is therefore represented by a Term-Document matrix,
noted $\mathbf{D}=[\mathbf{d}_1 \ \mathbf{d}_2 \ \dots \ \mathbf{d}_{m-1} \ \mathbf{d}_m]$
- The “nature” of word: will be discussed later
- From this basic representation, we will apply a sequence of successive embeddings, resulting in a global (valid) kernel with all desired properties

BOW kernels (II)

- Properties:

- All order information is lost (syntactical relationships, local context, ...)
- Feature space has dimension N (size of the dictionary)

- Similarity is basically defined by:

$$k(d_1, d_2) = \mathbf{d}_1 \cdot \mathbf{d}_2 = \mathbf{d}_1^t \cdot \mathbf{d}_2$$

or, normalized (cosine similarity):

$$\hat{k}(d_1, d_2) = \frac{k(d_1, d_2)}{\sqrt{k(d_1, d_1) \cdot k(d_2, d_2)}}$$

- Efficiency provided by sparsity (and sparse dot-product algo): $O(|d_1| + |d_2|)$

'Bag of words' kernels: enhancements

- The choice of indexing terms:
 - Exploit linguistic enhancements:
 - Lemma / Morpheme & stem
 - Disambiguated lemma (lemma+POS)
 - Noun Phrase (or useful collocation, n-grams)
 - Named entity (with type)
 - Grammatical dependencies (represented as feature vector components)
 - Ex: The human resource director of NavyCorp communicated important reports on ship reliability.
 - Exploit IR lessons
 - Stopword removal
 - Feature selection based on frequency
 - Weighting schemes (e.g. idf)
 - NB. Using polynomial kernels up to degree p , is a natural and efficient way of considering all (up-to-) p -grams (with different weights actually), but order is not taken into account (“sinking ships” is the same as “shipping sinks”)

'Bag of words' kernels: enhancements

■ Weighting scheme :

- the traditional *idf* weighting scheme
 $tf_i \rightarrow tf_i * \log(N/n_i)$
- is a linear transformation (scaling) $\phi(\mathbf{d}) \rightarrow \mathbf{W} \cdot \phi(\mathbf{d})$
 (where \mathbf{W} is diagonal): $k(d_1, d_2) = \phi(\mathbf{d}_1)^t \cdot (\mathbf{W}^t \cdot \mathbf{W}) \cdot \phi(\mathbf{d}_2)$
 can still be efficiently computed ($O(|d_1| + |d_2|)$)

■ Semantic expansion (e.g. synonyms)

- Assume some term-term similarity matrix \mathbf{Q}
 (positive definite) : $k(d_1, d_2) = \phi(\mathbf{d}_1)^t \cdot \mathbf{Q} \cdot \phi(\mathbf{d}_2)$
- In general, no sparsity (\mathbf{Q} : propagates)
- How to choose \mathbf{Q} (some kernel matrix for term)?

Semantic Smoothing Kernels

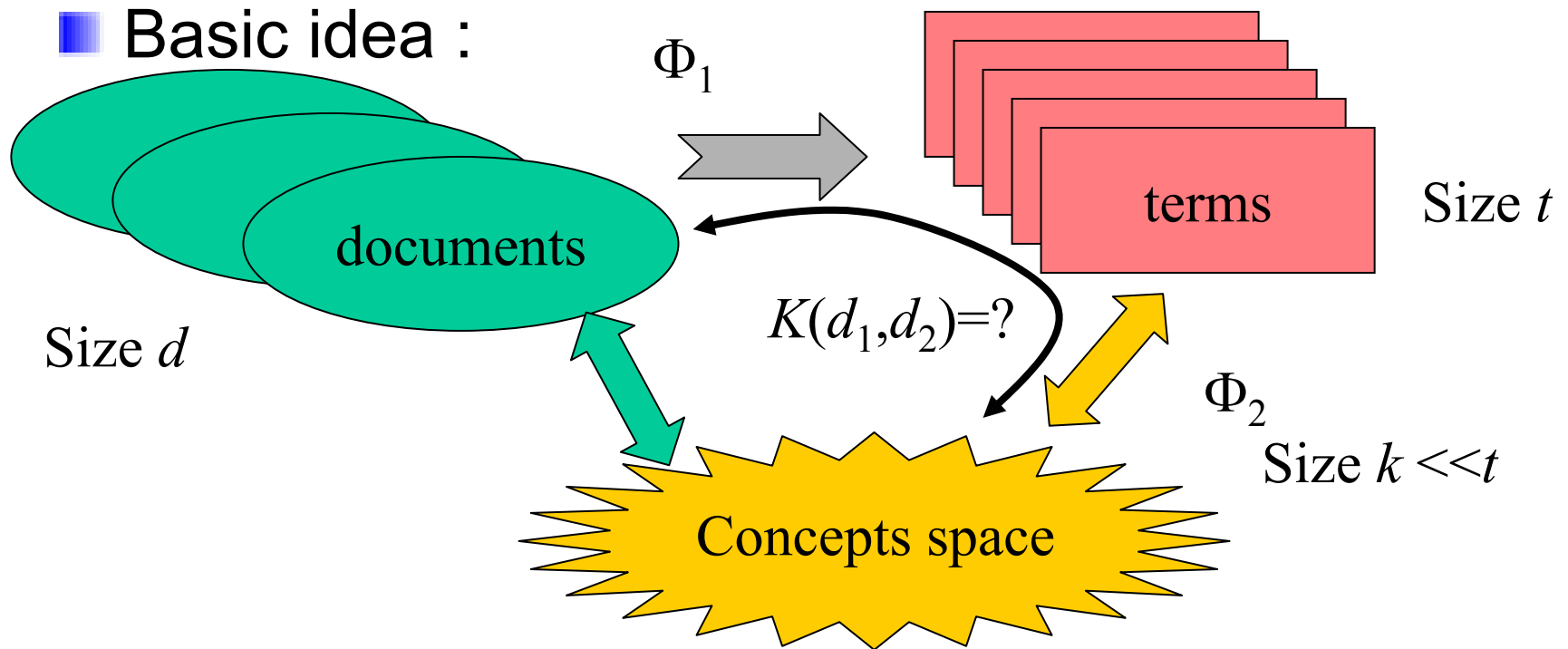
■ Synonymy and other term relationships:

- GVSM Kernel: the term-term co-occurrence matrix ($\mathbf{D}\mathbf{D}^t$) is used in the kernel: $k(d_1, d_2) = \mathbf{d}_1^t \cdot (\mathbf{D} \cdot \mathbf{D}^t) \cdot \mathbf{d}_2$
- The completely kernelized version of GVSM is:
 - The training kernel matrix $\mathbf{K} (= \mathbf{D}^t \cdot \mathbf{D}) \rightarrow \mathbf{K}^2$ ($m \times m$)
 - The kernel vector of a new document d vs the training documents : $\mathbf{t} \rightarrow \mathbf{K} \cdot \mathbf{t}$ ($m \times 1$)
 - The initial \mathbf{K} could be a polynomial kernel (GVSM on multi-words terms)
- Variants: One can use
 - a shorter context than the document to compute term-term similarity (term-context matrix)
 - Another measure than the number of co-occurrences to compute the similarity (e.g. Mutual information, ...)
- Can be generalised to \mathbf{K}^n (or a weighted combination of \mathbf{K}^1 \mathbf{K}^2 ... \mathbf{K}^n cfr. Diffusion kernels later), but is \mathbf{K}^n less and less sparse! Interpretation as sum over paths of length $2n$.

Semantic Smoothing Kernels

- Can use other term-term similarity matrix than DD^t ; e.g. a similarity matrix derived from the Wordnet thesaurus, where the similarity between two terms is defined as:
 - the inverse of the length of the path connecting the two terms in the hierarchical hyper/hyponymy tree.
 - A similarity measure for nodes on a tree (feature space indexed by each node n of the tree, with $\phi_n(\text{term } x)$ if term x is the class represented by n or “under” n), so that the similarity is the number of common ancestors (including the node of the class itself).
- With semantic smoothing, 2 documents can be similar even if they don't share common words.

Latent concept Kernels

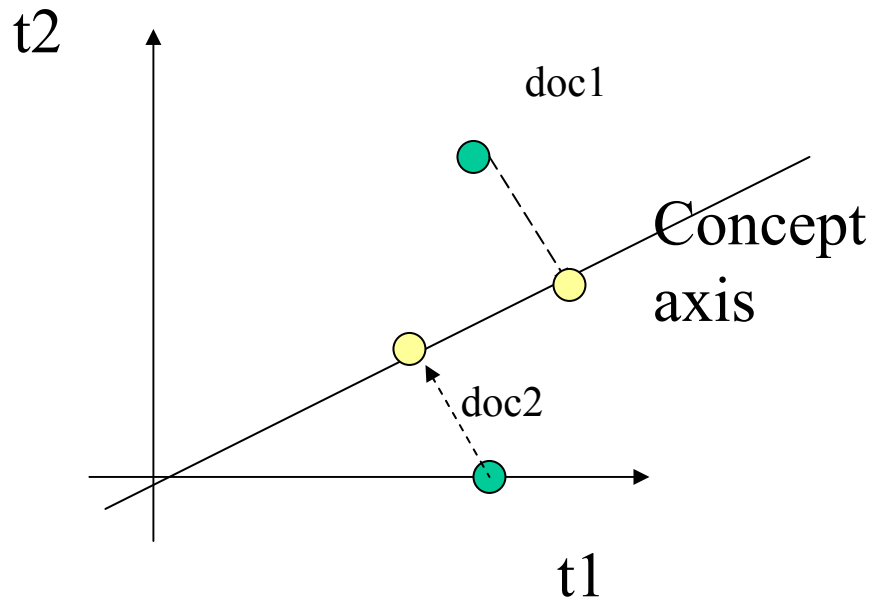


Latent concept Kernels

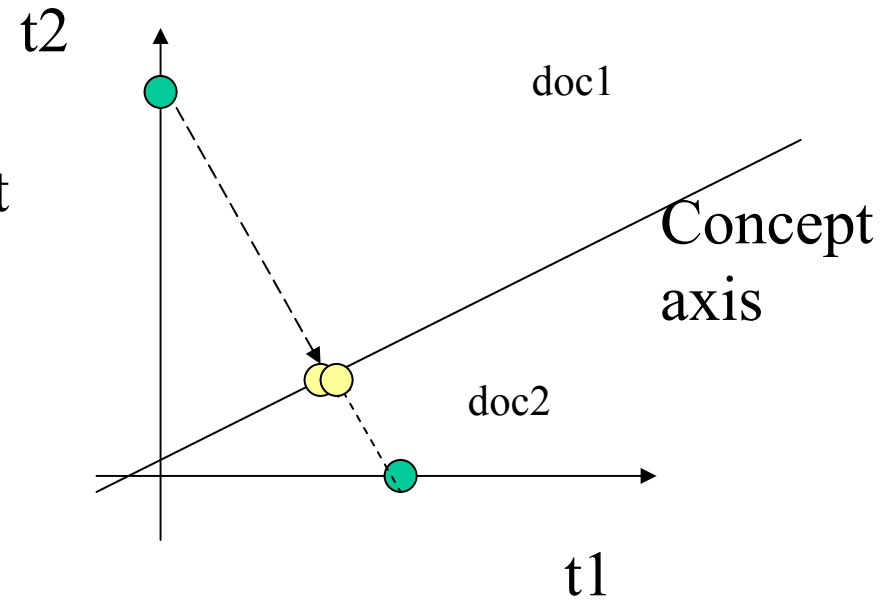
- $k(d_1, d_2) = \phi(d_1)^t \cdot \mathbf{P}' \cdot \mathbf{P} \cdot \phi(d_2)$,
 - where \mathbf{P} is a (linear) projection operator
 - From Term Space
 - to Concept Space
- Working with (latent) concepts provides:
 - Robustness to polysemy, synonymy, style, ...
 - Cross-lingual bridge
 - Natural Dimension Reduction
- But, how to choose \mathbf{P} and how to define (extract) the latent concept space? Ex: Use PCA : the concepts are nothing else than the principal components.

Polysemy and Synonymy

polysemy



synonymy



More formally ...

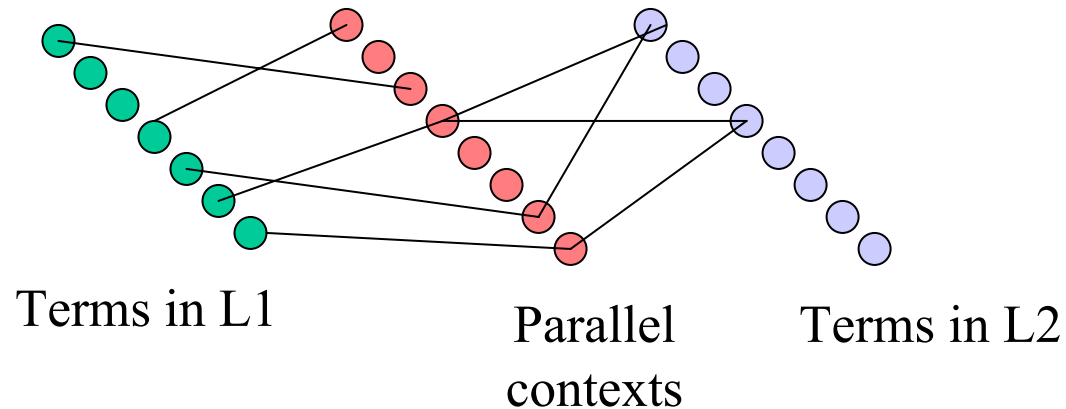
- SVD Decomposition of $\mathbf{D} \cong \mathbf{U}(t \times k) \mathbf{S}(k \times k) \mathbf{V}^t(k \times d)$, where \mathbf{U} and \mathbf{V} are projection matrices (from term to concept and from concept to document)
- Kernel Latent Semantic Indexing (SVD decomposition in feature space) :
 - \mathbf{U} is formed by the eigenvectors corresponding to the k largest eigenvalues of $\mathbf{D} \cdot \mathbf{D}^t$ (each column defines a concept by linear combination of terms)
 - \mathbf{V} is formed by the eigenvectors corresponding to the k largest eigenvalues of $\mathbf{K} = \mathbf{D}^t \mathbf{D}$
 - $\mathbf{S} = \text{diag}(\sigma_i)$ where σ_i^2 ($i=1, \dots, k$) is the i th largest eigenvalue of \mathbf{K}
 - Cfr semantic smoothing with $\mathbf{D} \cdot \mathbf{D}^t$ replaced $\mathbf{U} \cdot \mathbf{U}^t$ (new term-term similarity matrix): $k(d_1, d_2) = \mathbf{d}_1^t \cdot (\mathbf{U} \cdot \mathbf{U}^t) \cdot \mathbf{d}_2$
 - As in Kernel GVSM, the completely kernelized version of LSI is: $\mathbf{K} \rightarrow \mathbf{V} \mathbf{S}^2 \mathbf{V}'$ ($=\mathbf{K}$'s approximation of rank k) and $\mathbf{t} \rightarrow \mathbf{V} \mathbf{V}' \mathbf{t}$ (vector of similarities of a new doc), with no computation in the feature space
 - If $k \rightarrow n$, then latent semantic kernel is identical to the initial kernel

Complementary remarks

- Composition Polynomial kernel + kernel LSI (disjunctive normal form) or Kernel LSI + polynomial kernel (tuples of concepts or conjunctive normal form)
- GVSM is a particular case with one document = one concept
- Other decomposition :
 - Random mapping and randomized kernels (Monte-Carlo following some non-uniform distribution; bounds exist to probabilistically ensure that the estimated Gram matrix is ε -accurate)
 - Nonnegative matrix factorization $\mathbf{D} \cong \mathbf{A}(txk)\mathbf{S}(kxd)$ [$\mathbf{A} > 0$]
 - ICA factorization $\mathbf{D} \cong \mathbf{A}(txk)\mathbf{S}(kxd)$ (kernel ICA)
 - Cfr semantic smoothing with $\mathbf{D}.\mathbf{D}^t$ replaced by $\mathbf{A}.\mathbf{A}^t$:
 $k(d_1, d_2) = \mathbf{d}_1^t . (\mathbf{A}.\mathbf{A}^t) . \mathbf{d}_2$
 - Decompositions coming from multilingual parallel corpora (crosslingual GVSM, crosslingual LSI, CCA)

Why multilingualism helps ...

■ Graphically:



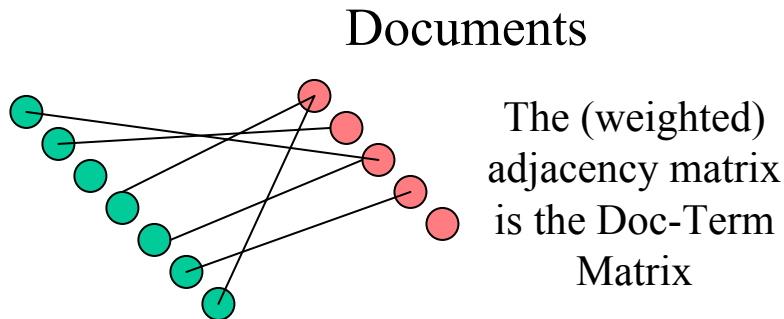
- Concatenating both representations will force language-independent concept: each language imposes constraints on the other
- Searching for maximally correlated projections of paired observations (CCA) has a sense, semantically speaking

Diffusion Kernels

- Recursive dual definition of the semantic smoothing:
 - $\mathbf{K} = \mathbf{D}'(\mathbf{I} + u\mathbf{Q})\mathbf{D}$
 - $\mathbf{Q} = \mathbf{D}(\mathbf{I} + v\mathbf{K})\mathbf{D}'$
 - NB. $u=v=0 \rightarrow$ standard BOW; $v=0 \rightarrow$ GVSM
- Let $\mathbf{B} = \mathbf{D}'\mathbf{D}$ (standard BOW kernel); $\mathbf{G} = \mathbf{D}\mathbf{D}'$
- If $u=v$, The solution is the “Von Neumann diffusion kernel”
 - $\mathbf{K} = \mathbf{B} \cdot (\mathbf{I} + u\mathbf{B} + u^2\mathbf{B}^2 + \dots) = \mathbf{B}(\mathbf{I} - u\mathbf{B})^{-1}$ and $\mathbf{Q} = \mathbf{G}(\mathbf{I} - u\mathbf{G})^{-1}$ [only of $u < \|\mathbf{B}\|^{-1}$]
 - Can be extended, with a faster decay, to exponential diffusion kernel: $\mathbf{K} = \mathbf{B} \cdot \exp(u\mathbf{B})$ and $\mathbf{Q} = \exp(u\mathbf{G})$

Graphical Interpretation

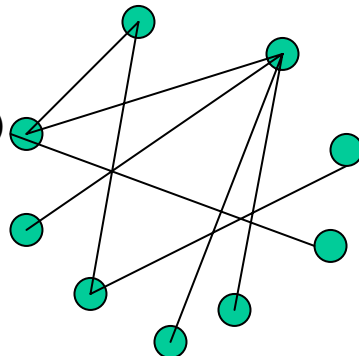
- These diffusion kernels correspond to defining similarities between nodes in a graph



Terms

Or

By aggregation, the (weighted) adjacency matrix is the term-term similarity matrix \mathbf{G}



Diffusion kernels corresponds to considering all paths of length 1, 2, 3, 4 ... linking 2 nodes and summing the product of local similarities, with different decay strategies

It is in some way similar to KPCA by just “rescaling” the eigenvalues of the basic Kernel matrix (decreasing the lowest ones)

Strategies of Design

- Kernel as a way to encode prior information
 - Invariance: synonymy, document length, ...
 - Linguistic processing: word normalisation, semantics, stopwords, weighting scheme, ...
- **Convolution Kernels:** text is a recursively-defined data structure. How to build “global” kernels from local (atomic level) kernels?
- Generative model-based kernels: the “topology” of the problem will be translated into a kernel function

Sequence kernels

- Consider a document as:
 - A sequence of characters (string)
 - A sequence of tokens (or stems or lemmas)
 - A paired sequence (POS+lemma)
 - A sequence of concepts
 - A tree (parsing tree)
 - A dependency graph
- }

(later)
- Sequence kernels → order has importance
 - Kernels on string/sequence : counting the subsequences two objects have in common ... but various ways of counting
 - Contiguity is necessary (p-spectrum kernels)
 - Contiguity is not necessary (subsequence kernels)
 - Contiguity is penalised (gap-weighted subsequence kernels)

String and Sequence

- Just a matter of convention:
 - String matching: implies contiguity
 - Sequence matching : only implies order

p -spectrum kernel

- Features of $s = p$ -spectrum of $s =$ histogram of all (contiguous) substrings of length p
- Feature space indexed by all elements of Σ^p
- $\phi_u(s)$ =number of occurrences of u in s
- Ex:
 - s ="John loves Mary Smith"
 - t ="Mary Smith loves John"

	JL	LM	MS	SL	LJ
s	1	1	1		
t			1	1	1

$$K(s,t)=1$$

p-spectrum Kernels (II)

■ Naïve implementation:

- For all p -grams of s , compare equality with the p -grams of t
- $O(p|s||t|)$
- Later, implementation in $O(p(|s|+|t|))$

All-subsequences kernels

- Feature space indexed by all elements of $\Sigma^* = \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \dots$
- $\phi_u(s)$ = number of occurrences of u as a (non-contiguous) subsequence of s
- Explicit computation rapidly infeasible (exponential in $|s|$ even with sparse rep.)

■ Ex:

	ε	J	L	M	S	J L	L M	M S	S L	L J	J L M	L M S	M S L	S L J	J L M S	M S L J
s	1	1	1	1	1	1	1	1			1	1			1	
t	1	1	1	1	1			1	1	1			1	1		1

K=6

Recursive implementation

- Consider the addition of one extra symbol a to s : common subsequences of (sa, t) are either in s or must end with symbol a (in both sa and t).
- Mathematically,

$$k(s, \varepsilon) = 1$$

$$k(sa, t) = k(s, t) + \sum_{j: t_j = a} k(s, t(1:j-1)) = k(s, t) + k'(sa, t)$$

$$k'(sa, tv) = k'(sa, t) + k(s, t)\delta_{va}$$

- This gives a complexity of $O(|s||t|)$

Practical implementation (DP table)

s\t	ϵ	John	admires	Mary	Ann	Smith
ϵ	1	1	1	1	1	1
K'(jonn)	0	1	1	1	1	1
John	1	2	2	2	2	2
K'(loves)	0	0	0	0	0	0
Loves	1	2	2	2	2	2
K'(Mary)	0	0	0	2	2	2
Mary	1	2	2	4	4	4

NB: by-product : all $k(a,b)$ for prefixes a of s , b of t

Fixed-length subsequence kernels

- Feature space indexed by all elements of Σ^p
- $\phi_u(s)$ = number of occurrences of the p -gram u as a (non-contiguous) subsequence of s
- Recursive implementation (will create a series of p tables)

$$k_0(s, \varepsilon) = 1$$

$$k_p(s, \varepsilon) = 0 \quad \forall p > 0$$

$$k_p(sa, t) = k_p(s, t) + \sum_{j: t_j = a} k_{p-1}(s, t(1:j-1)) = k(s, t) + k_p'(sa, t)$$

$$k_p'(sa, tv) = k_{p-1}'(sa, t) + k_{p-1}(s, t) \delta_{va}$$

- Complexity: $O(p|s||t|)$, but we have the k -length subseq. kernels ($k \leq p$) for free \rightarrow easy to compute $k(s, t) = \sum a_i k_i(s, t)$

Gap-weighted subsequence kernels

- Feature space indexed by all elements of Σ^p
- $\phi_u(s)$ = sum of weights of occurrences of the p-gram u as a (non-contiguous) subsequence of s , the weight being length penalizing: $\lambda^{\text{length}(u)}$ [NB: length includes both matching symbols and gaps]
- Example:
 - D1 : ATCGTAGACTIGTC
 - D2 : GACTATGC
 - $(D1)_{\text{CAT}} = 2\lambda^8 + 2\lambda^{10}$ and $(D2)_{\text{CAT}} = \lambda^4$
 - $k(D1, D2)_{\text{CAT}} = 2\lambda^{12} + 2\lambda^{14}$
- Naturally built as a dot product \rightarrow valid kernel
- For alphabet of size 80, there are 512000 trigrams
- For alphabet of size 26, there are $12 \cdot 10^6$ 5-grams

Gap-weighted subsequence kernels

- Hard to perform explicit expansion and dot-product!
- Efficient recursive formulation (dynamic programming –like), whose complexity is $O(k \cdot |D1| \cdot |D2|)$
- Normalization (doc length independence)

$$\hat{k}(d_1, d_2) = \frac{k(d_1, d_2)}{\sqrt{k(d_1, d_1) \cdot k(d_2, d_2)}}$$

Recursive implementation

- Defining $K'_i(s,t)$ as $K_i(s,t)$, but the occurrences are weighted by the length to the end of the string (s)

$$K_i(sa, t) = K_i(s, t) + \sum_{j:t_j=a} \lambda^2 K'_{i-1}(s, t[1:j-1])$$

$$K'_i(sa, t) = \lambda K'_i(s, t) + \sum_{j:t_j=a} \lambda^{|t|-j+1} K'_{i-1}(s, t[1:j-1]) = \lambda K'_i(s, t) + K''_i(sa, t)$$

$$K''_i(sa, tv) = K''_{i-1}(sa, t) \cdot \lambda + K'_{i-1}(sa, tv) \cdot \delta_{av} \cdot \lambda^2$$

$$K'_0(s, t) = 1$$

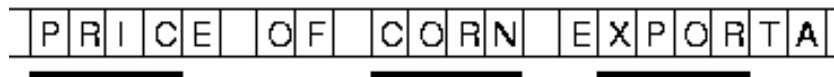
$$K'_i(s, t) = 0 \text{ if } \min(|s|, |t|) < i$$

- 3.p DP tables must be built and maintained
- As before, as by-product, all gap-weighted k-grams kernels, with $k \leq p$ so that any linear combination $k(s,t) = \sum a_i k_i(s,t)$ easy to compute

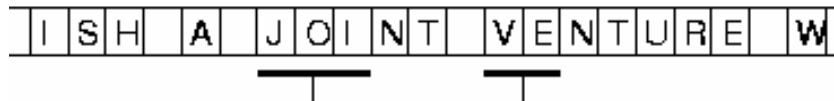
Word Sequence Kernels (I)

- Here “words” are considered as symbols
 - Meaningful symbols → more relevant matching
 - Linguistic preprocessing can be applied to improve performance
 - Shorter sequence sizes → improved computation time
 - But increased sparsity (documents are more : “orthogonal”)
- Motivation : the noisy stemming hypothesis (important N-grams approximate stems), confirmed experimentally in a categorization task

4-GRAMS



5-GRAMS



Word Sequence Kernels (II)

- Link between Word Sequence Kernels and other methods:
 - For $k=1$, WSK is equivalent to basic “Bag Of Words” approach
 - For $\lambda=1$, close relation to polynomial kernel of degree k , WSK takes order into account
- Extension of WSK:
 - Symbol dependant decay factors (way to introduce IDF concept, dependence on the POS, stop words)
 - Different decay factors for gaps and matches (e.g. $\lambda_{\text{noun}} < \lambda_{\text{adj}}$ when gap; $\lambda_{\text{noun}} > \lambda_{\text{adj}}$ when match)
 - Soft matching of symbols (e.g. based on thesaurus, or on dictionary if we want cross-lingual kernels)

Recursive equations for variants

- It is obvious to adapt recursive equations, without increasing complexity

$$K_i(sa, t) = K_i(s, t) + \sum_{j:t_j=a} \lambda_{a, \text{match}}^2 K'_{i-1}(s, t[1:j-1])$$

$$K'_i(sa, t) = K'_i(s, t) \cdot \lambda_{a, \text{gap}} + K''_i(sa, t)$$

$$K''_i(sa, tv) = K''_{i-1}(sa, t) \cdot \lambda_{v, \text{gap}} + K'_{i-1}(sa, tv) \cdot \delta_{av} \cdot \lambda_{v, \text{match}}^2$$

Or, for soft matching ($b_{i,j}$: elementary symbol kernel):

$$K_i(sa, t) = K_i(s, t) + \sum_{j=i}^{|t|} K'_{i-1}(s, t[1:j-1]) \lambda^2 b_{a,t_j}$$

$$K'_i(sa, t) = K'_i(s, t) \cdot \lambda + K''_i(sa, t)$$

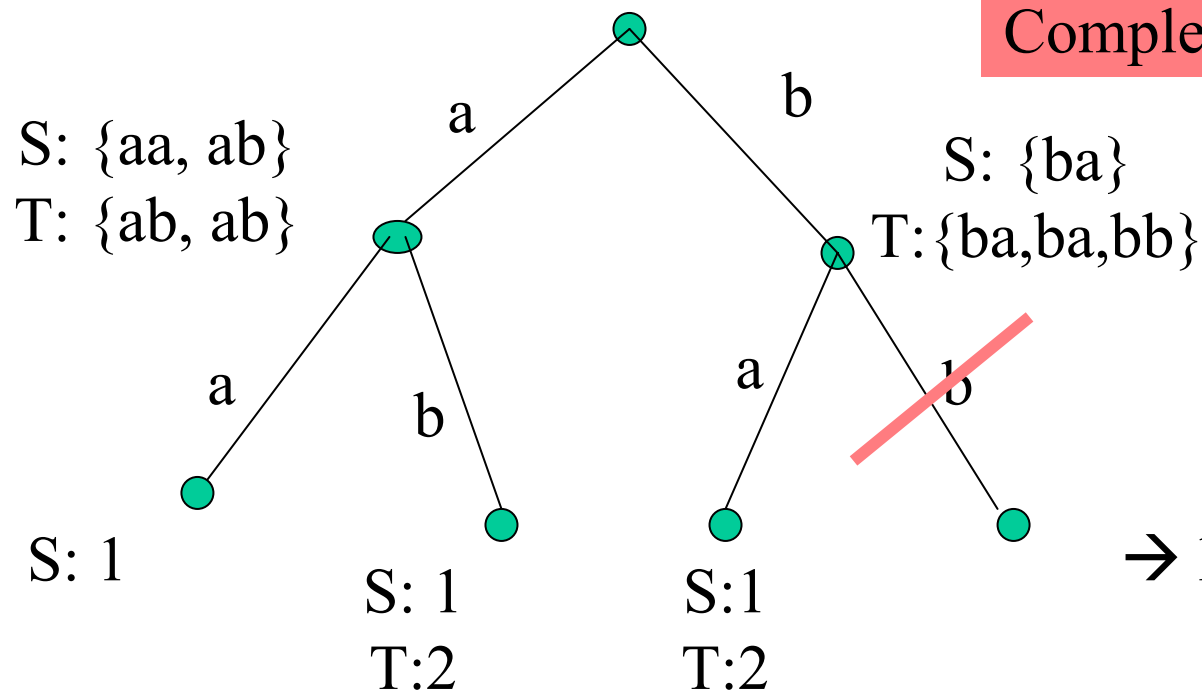
$$K''_i(sa, tv) = K''_{i-1}(sa, t) \cdot \lambda + K'_{i-1}(sa, tv) \cdot b_{a,v} \cdot \lambda^2$$

Trie-based kernels

- An alternative to DP based on string matching techniques
- TRIE= Retrieval Tree (cfr. Prefix tree) = tree whose internal nodes have their children indexed by Σ .
- Suppose $F = \Sigma^p$: the leaves of a complete p -trie are the indices of the feature space
- Basic algorithm:
 - Generate all substrings $s(i:j)$ satisfying initial criteria; idem for t .
 - Distribute the s -associated list down from root to leaf
 - Distribute the t -associated list down from root to leaf (breadth-first), taking into account the distribution of s -list (pruning)
 - Compute the product at the leaves and sum over the leaves
- Key points: in steps (2) and (3), not all the leaves will be populated (else complexity would be $O(|\Sigma^p|)$)

Example 1 – p-spectrum

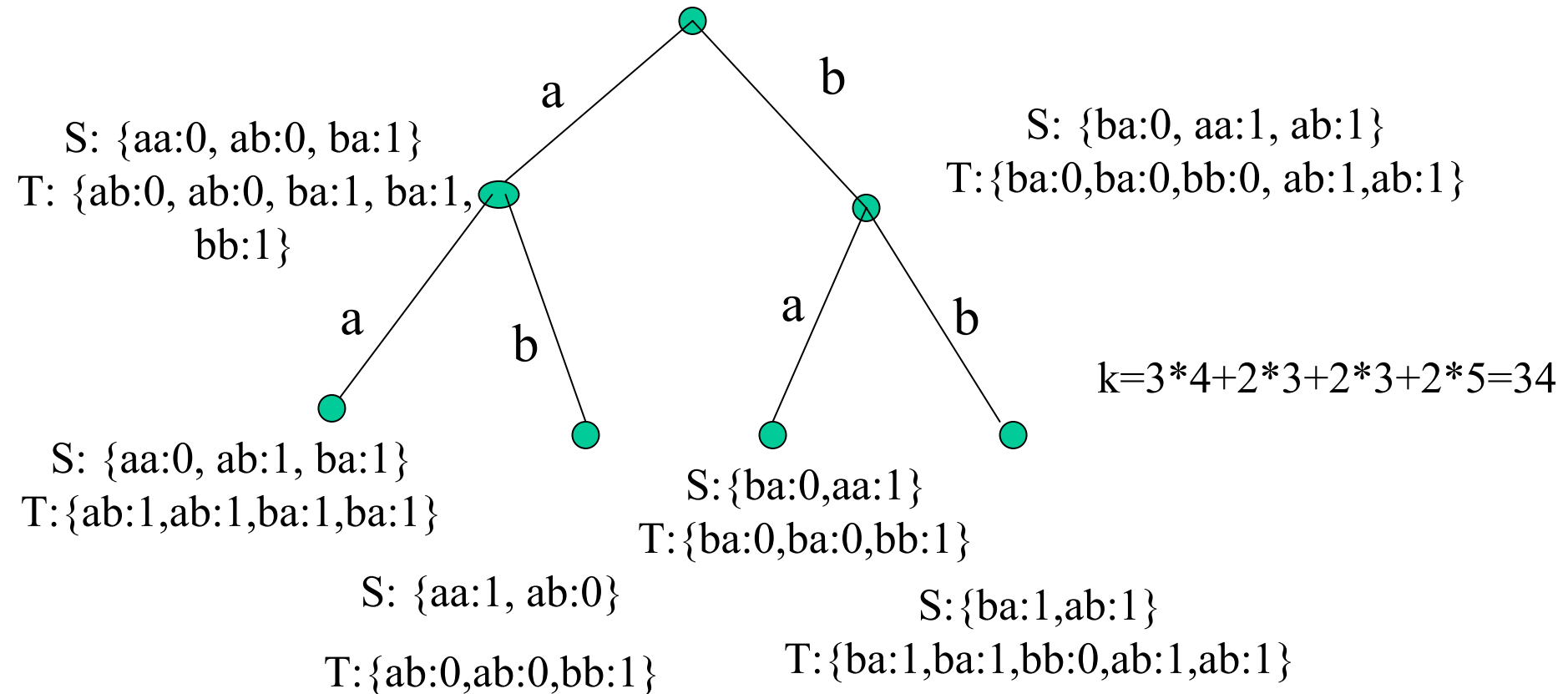
- $p=2$
- $S = a a b a \rightarrow \{aa, ab, ba\}$
- $T = b a b a b b \rightarrow \{ba, ab, ba, ab, bb\}$



Example 2: (p,m) -mismatch kernels

- Feature space indexed by all elements of Σ^p
- $\phi_u(s)$ =number of p -grams (substrings) of s that differ from u by at most m symbols
- See example on next slide ($p=2; m=1$)
- Complexity $O(p^{m+1}|\Sigma|^m(|s|+|t|))$
- Can be easily extended by using a semantic (local) dissimilarity matrix $b_{a,b}$, to
 $\phi_u(s)$ =number of p -grams (substrings) of s that differ from u by a total dissimilarity not larger than some threshold (total=sum)

Example 2: illustration



Example 3: restricted gap-weighted kernel

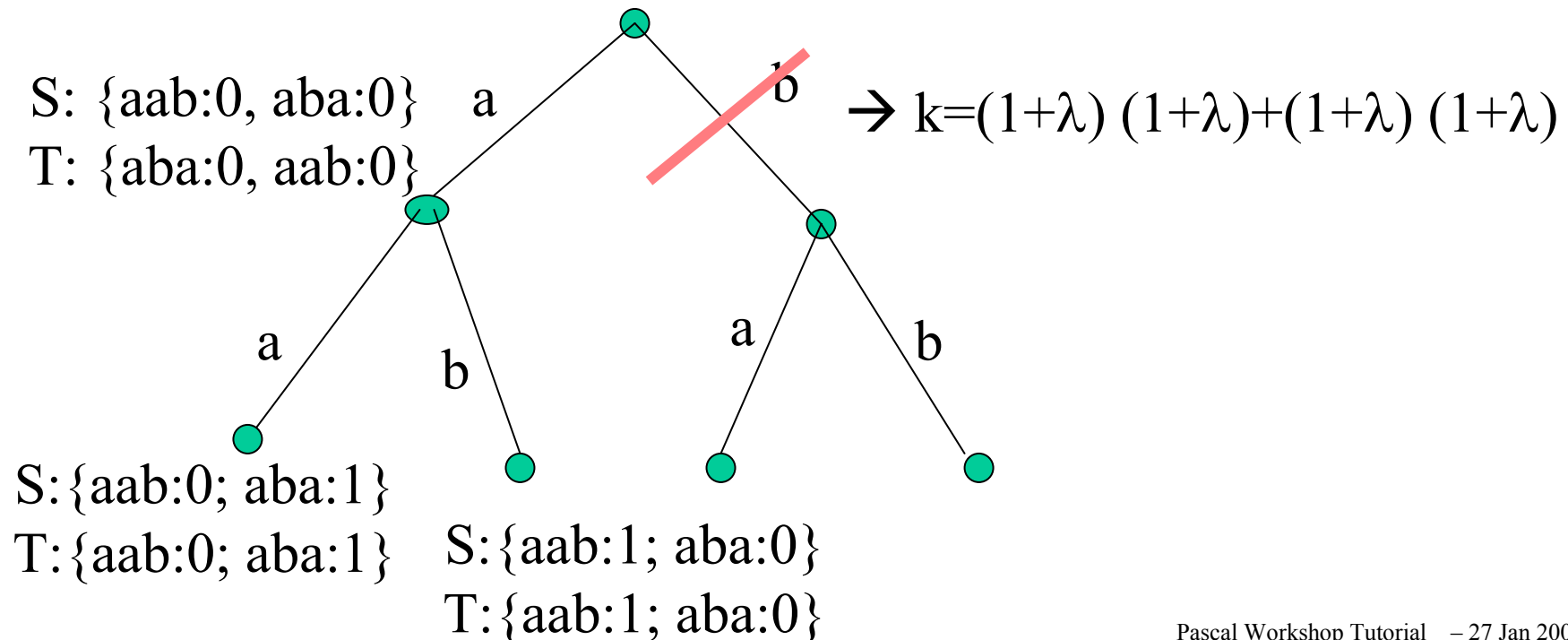
- Feature space indexed by all elements of Σ^p
- $\phi_u(s)$ = sum of weights of occurrences of p -gram u as a (non-contiguous) subsequence of s , provided that u occurs with at most m gaps, the weight being gap penalizing: $\lambda^{\text{gap}(u)}$
- For small λ , restrict m to 2 or even 1 is a reasonable approximation of the full gap-weighted subsequence kernel
- Cfr. Previous algorithms but generate all $(p+m)$ substrings at the initial phase.
- Complexity $O((p+m)^m(|s|+|t|))$
- If m is too large, DP is more efficient

Example 3 : illustration

■ $p=2 ; m=1$

■ $S=a a b a \rightarrow \{aab, aba\}$

■ $T=b a b a a b \rightarrow \{bab, aba, baa, aab\}$

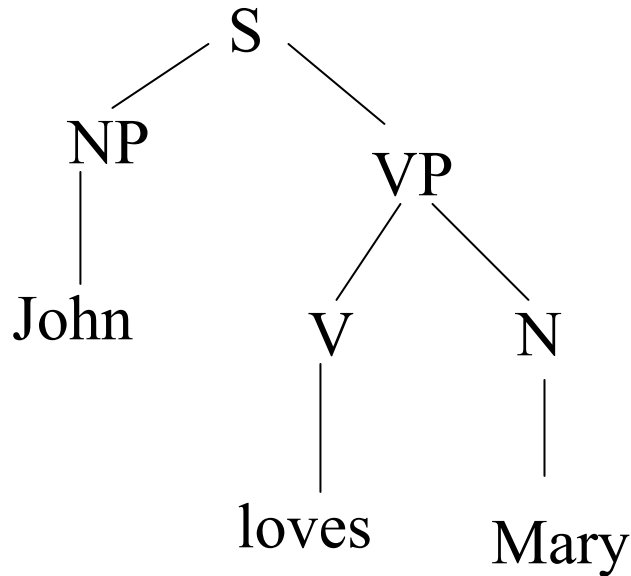


Tree Kernels

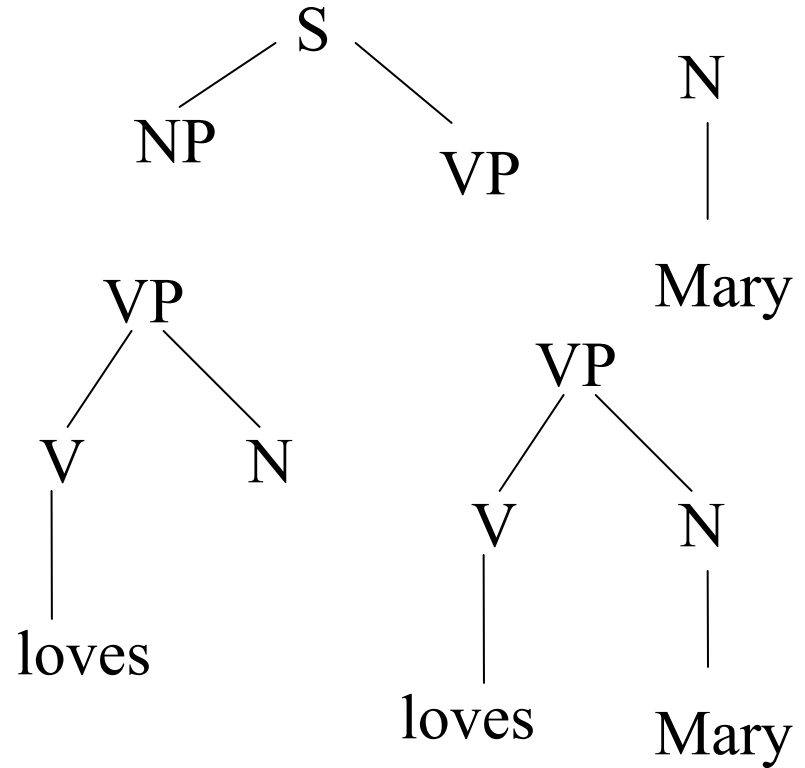
- Application: categorization [one doc=one tree], parsing (desambiguation) [one doc = multiple trees]
- Tree kernels constitute a particular case of more general kernels defined on discrete structure (convolution kernels). Intuitively, the philosophy is
 - to split the structured objects in parts,
 - to define a kernel on the “atoms” and a way to recursively combine kernel over parts to get the kernel over the whole.
- Feature space definition: one feature for each possible proper subtree in the training data; feature value = number of occurrences
- A subtree is defined as any part of the tree which includes more than one node, with the restriction there is no “partial” rule production allowed.

Tree Kernels : example

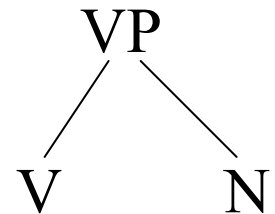
■ Example :



A Parse Tree



... a few among
the many subtrees
of this tree!



Tree Kernels : algorithm

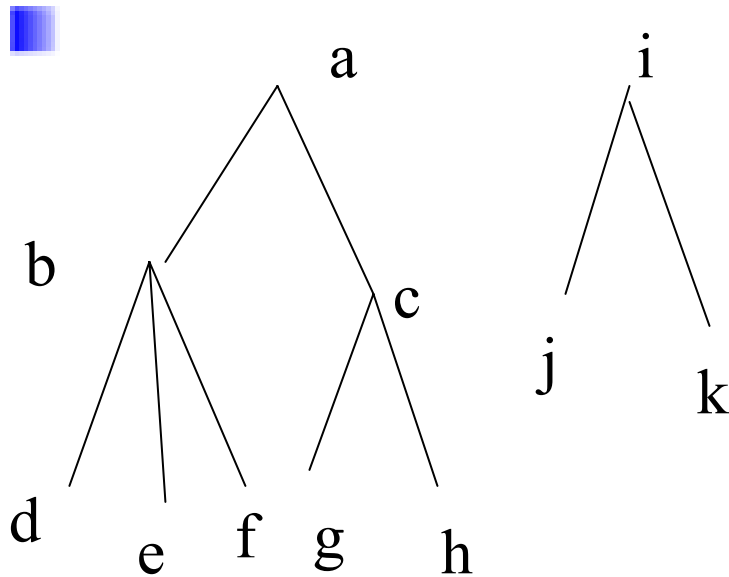
- Kernel = dot product in this high dimensional feature space
- Once again, there is an efficient recursive algorithm (in polynomial time, not exponential!)
- Basically, it compares the production of all possible pairs of nodes (n_1, n_2) ($n_1 \in T_1, n_2 \in T_2$); if the production is the same, the number of common subtrees rooted at both n_1 and n_2 is computed recursively, considering the number of common subtrees rooted at the common children
- Formally, let $k_{\text{co-rooted}}(n_1, n_2)$ = number of common subtrees rooted at both n_1 and n_2

$$k(T_1, T_2) = \sum_{n_1 \in T_1} \sum_{n_2 \in T_2} k_{\text{co-rooted}}(n_1, n_2)$$

All sub-tree kernel

- $K_{\text{co-rooted}}(n_1, n_2) = 0$ if n_1 or n_2 is a leaf
- $K_{\text{co-rooted}}(n_1, n_2) = 0$ if n_1 and n_2 have different production or, if labeled, different label
- Else $K_{\text{co-rooted}}(n_1, n_2) = \prod_{\text{children } i} (1 + k_{\text{co-rooted}}(\text{ch}(n_1, i), \text{ch}(n_2, i)))$
- “Production” is left intentionally ambiguous to both include unlabelled tree and labeled tree
- Complexity is $O(|T_1| \cdot |T_2|)$

Illustration



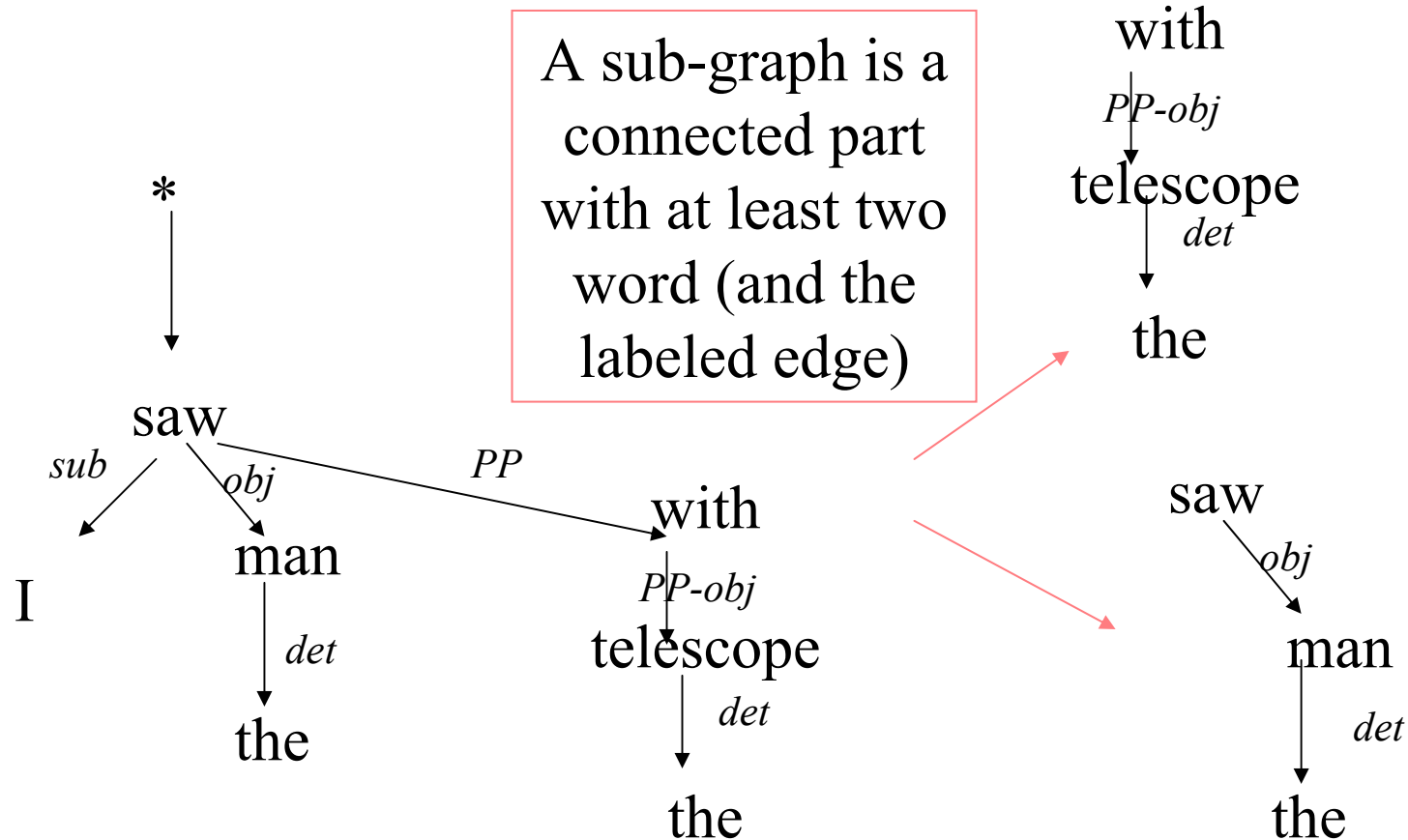
Kco-root	i	j	k
a	1	0	0
b	0	0	0
c	1	0	0
d	0	0	0
e	0	0	0
f	0	0	0
g	0	0	0
h	0	0	0

→K=2

Tree kernels : remarks

- Normalisation: downweighting larger structure feature by a decay factor $\lambda^{\text{size}(\text{subtree})}$
- Similar definitions and algorithms for:
 - Dependency graphs (labeled nodes [word] AND edges [syntactical relationship])
 - Paired sequences (e.g. the words of a sentence and the corresponding POS)

Dependency Graph Kernel

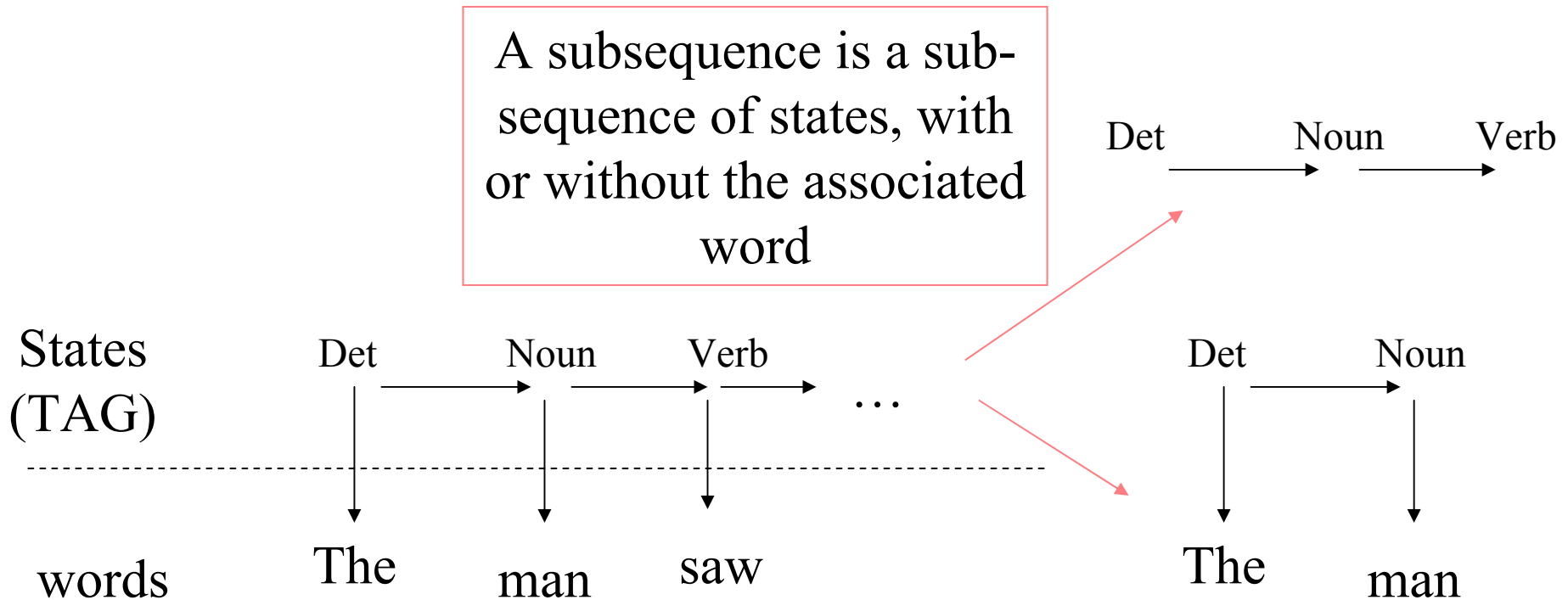


Dependency Graph Kernel

$$k(D_1, D_2) = \sum_{n_1 \in D_1} \sum_{n_2 \in D_2} k_{co-rooted}(n_1, n_2)$$

- $K_{co-rooted}(n_1, n_2) = 0$ if n_1 or n_2 has no child
- $K_{co-rooted}(n_1, n_2) = 0$ if n_1 and n_2 have different label
- Else $K_{co-rooted}(n_1, n_2) = \prod_{x, y \in \text{common dependencies}} (2 + k_{co-rooted}(x, y)) - 1$

Paired sequence kernel



Paired sequence kernel

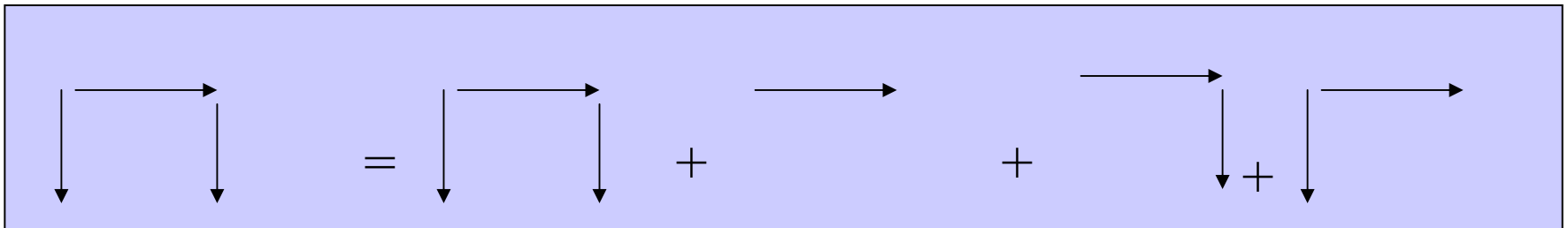
$$k(S_1, S_2) = \sum_{n_1 \in \text{states}(S_1)} \sum_{n_2 \in \text{states}(D_2)} k_{\text{co-rooted}}(n_1, n_2)$$

■ $K_{\text{co-rooted}}(n_1, n_2) = 0$ if n_1 and n_2 have different tags

■ Else, if n_1 and n_2 have different words,

$$K_{\text{co-rooted}}(n_1, n_2) = K_{\text{co-rooted}}(\text{next}(n_1), \text{next}(n_2))$$

■ Else $K_{\text{co-rooted}}(n_1, n_2) = 1 + 2 * K_{\text{co-rooted}}(\text{next}(n_1), \text{next}(n_2))$



Strategies of Design

- Kernel as a way to encode prior information
 - Invariance: synonymy, document length, ...
 - Linguistic processing: word normalisation, semantics, stopwords, weighting scheme, ...
- Convolution Kernels: text is a recursively-defined data structure. How to build “global” kernels from local (atomic level) kernels?
- **Generative model-based kernels: the “topology” of the problem will be translated into a kernel function**

Marginalised – Conditional Independence Kernels

- Assume a family of models M (with prior $p_0(m)$ on each model) [finite or countably infinite]
- each model m gives $P(x|m)$
- Feature space indexed by models: $x \rightarrow P(x|m)$
- Then, assuming conditional independence, the joint probability is given by

$$P_M(x, z) = \sum_{m \in M} P(x, z | m) P_0(m) = \sum_{m \in M} P(x | m) P(z | m) P_0(m)$$

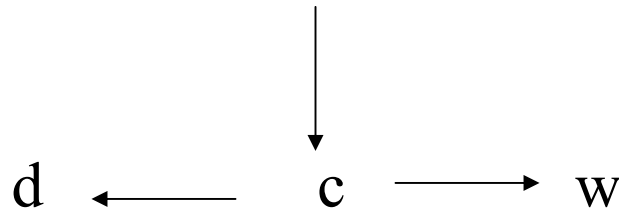
- This defines a valid probability-kernel (CI implies PD kernel), by marginalising over m . Indeed, the gram matrix is $K = P \cdot \text{diag}(P_0) \cdot P'$ (some reminiscence of latent concept kernels)

Remind

- This family of strategies brings you the additional advantage of using all your unlabeled training data to design more problem-adapted kernels
- They constitute a natural and elegant way of solving semi-supervised problems (mix of labelled and unlabelled data)

Exemple 1: PLSA-kernel (somewhat artificial)

- Probabilistic Latent Semantic Analysis provides a generative model of both documents and words in a corpus:

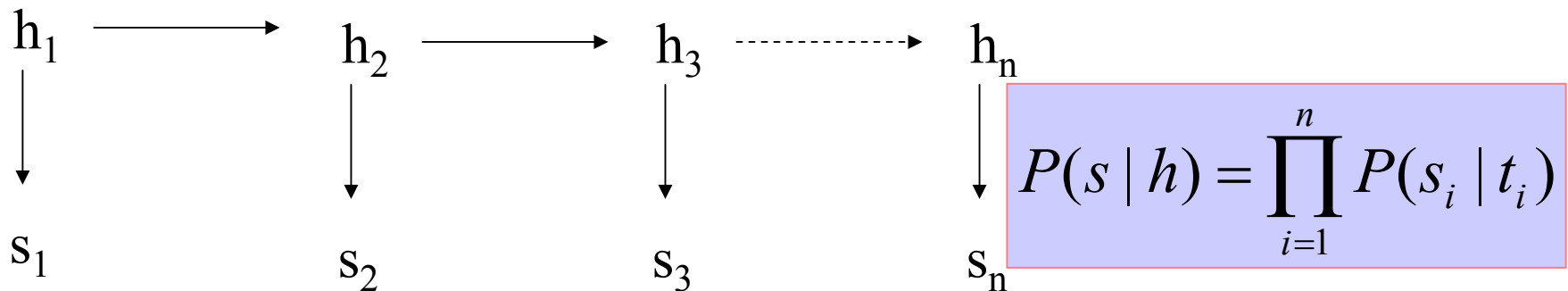


- $P(d,w)=\sum P(c)P(d|c)P(w|c)$
- Assuming that the topics is the model, you can identify the models $P(d|c)$ and $P(c)$
- Then use marginalised kernels

$$P_M(d_1, d_2) = \sum_{c \in M} P(d_1 | c) P(d_2 | c) P_0(c)$$

Example 2: HMM generating fixed-length strings

- The generative models of a string s (of length n) is given by an HMM, represented by (A is the set of states)



- Then $k(s, t) = \sum_{\text{path } h \in A^n} \prod_{i=1}^n P(s_i | h_i) \cdot P(t_i | h_i) \cdot P(h_i | h_{i-1})$

- With efficient recursive (DP) implementation in $O(n|A|^2)$

Fisher Kernels

- Assume you have only 1 model
 - Marginalised kernel give you little information: only one feature: $P(x|m)$
 - To exploit much, the model must be “flexible”, so that we can measure how it adapts to individual items → we require a “smoothly” parametrised model
 - Link with previous approach: locally perturbed models constitute our family of models, but $\dim F = \text{number of parameters}$
- More formally, let $P(x|\theta_0)$ be the generative model (θ_0 is typically found by max likelihood); the gradient $\nabla_{\theta} \log P(x|\theta)|_{\theta=\theta_0}$ reflects how the model will be changed to accommodate the new point x (NB. In practice the loglikelihood is used)

Fisher Kernel : formally

- Two objects are similar if they require similar adaptation of the parameters or, in other words, if they stretch the model in the same direction:

$$K(x,y) = (\nabla_{\theta} \log P(x | \theta)|_{\theta=\theta_0})' I_M^{-1} (\nabla_{\theta} \log P(y | \theta)|_{\theta=\theta_0})$$

Where I_M is the Fisher Information Matrix

$$I_M = E[(\nabla_{\theta} \log P(x | \theta)|_{\theta=\theta_0})(\nabla_{\theta} \log P(x | \theta)|_{\theta=\theta_0})']$$

On the Fisher Information Matrix

- The FI matrix gives some non-euclidian, topological-dependant dot-product
- It also provides invariance to any smooth invertible reparametrisation
- Can be approximated by the empirical covariance on the training data points
- But, it can be shown that it increase the risk of amplifying noise if some parameters are not relevant
- Practically, I_M is taken as I.

Example 1 : Language model

- Language models can improve p-spectrum kernels
- Language model is a generative model: $p(w_n | w_{n-k} \dots w_{n-1})$ are the parameters
- The likelihood of $s = \prod_{j=1}^{n-k} p(s_{j+k} | s_j \dots s_{j+k-1})$
- The gradient of the log-likelihood with respect to parameter $u \rightarrow v$ is the number of occurrences of uv in s divided by $p(u \rightarrow v)$, minus the number of occurrences of u in s (this second term realises the projection of the gradient along the constraint $\sum p(u \rightarrow v) = 1$)

Example 2 : PLSA-Fisher Kernels

- An example : Fisher kernel for PLSA improves the standard BOW kernel

$$K(d_1, d_2) = \sum_c \frac{P(c|d_1) \cdot P(c|d_2)}{P(c)} + \sum_w \tilde{t}f(w, d_1) \tilde{t}f(w, d_2) \sum_c \frac{P(c|d_1, w) \cdot P(c|d_2, w)}{P(w|c)}$$

- where $k_1(d_1, d_2)$ is a measure of how much d_1 and d_2 share the same latent concepts (synonymy is taken into account)
- where $k_2(d_1, d_2)$ is the traditional inner product of common term frequencies, but weighted by the degree to which these terms belong to the same latent concept (polysemy is taken into account)

Applications of SVM in NLP

- Document categorization and filtering
- Event Detection and Tracking
- Chunking and Segmentation
- Disambiguation:
 - POS –tagging
 - Word sense disambiguation

Document Categorization & Filtering

- Classification task
 - Classes = topics
 - Or Class = relevant / not relevant (filtering)
- Typical corpus: 30.000 features , 10.000 training documents

Break-even point	Reuters	WebKb	Ohsumed
Naïve Bayes	72.3	82.0	62.4
Rocchio	79.9	74.1	61.5
C4.5	79.4	79.1	56.7
K-NN	82.6	80.5	63.4
SVM	87.5	90.3	71.6

- NB. David Lewis (this morning's tutorial!) won the TREC2001 batch filtering track using SVMs

SVM for Chunk Identification

- Each word has to be tagged with a chunk label (combination IB / chunk type). E.g. I-NP, B-NP, ...
- This can be seen as a classification problem with typically ± 20 categories (multiclass problem – solved by *one-vs-rest* or *pairwise* classification and *max* or *majority* voting - $K \times (K-1) / 2$ classifiers)
- Typical feature vector: surrounding context (word and POS-tag) and (estimated) chunk labels of previous words
- Results on CoNLL-00 corpus: 93.5%

SVM for Word Sense Disambiguation

- Can be considered as a classification task (choose between some predefined senses)
- NB. One (multiclass) classifier for each ambiguous word
- Typical Feature Vector:
 - Surrounding context (words and POS tags)
 - Presence/absence of focus-specific keywords in a wider context
- As usual in NLP problems, few training examples and many features
- On the 1998 Senseval competition Corpus, SVM has an average rank of 2.3 (in competition with 8 other learning algorithms on about 30 ambiguous words)

SVM for POS Tagging

- POS tagging is a multiclass classification problem
- Typical Feature Vector (for unknown words):
 - Surrounding context: words of both sides
 - Morphologic info: pre- and suffixes, existence of capitals, numerals, ...
 - POS tags of preceding words
- Results on the Penn Treebank WSJ corpus (training data: 1,000,000 tokens) for unknown words – Polynomial kernel ($d=2$) : accuracy = 87.1% (comparable with or better than TnT, 2nd order Markov Model)